An Automated Formal Approach to Managing Dynamic Reconfiguration

Ian Warren[1], Jing Sun[1], Sanjev Krishnamohan[2] and Thiranjith Weerasinghe[2]
Department of Computer Science[1], Department of Electrical and Computer Engineering[2]
The University of Auckland, Private Bag 92019, Auckland, New Zealand
{ian-w, j.sun}@cs.auckland.ac.nz, {skri018, twee003}@ec.auckland.ac.nz

Abstract

Dynamic reconfiguration is the process of making changes to software at run-time. The motivation for this is typically to facilitate adaptive systems which change their behavior in response to changes in their operating environment or to allow systems with a requirement for continuous service to evolve uninterrupted. To enable development of reconfigurable applications, we have developed OpenRec, a framework which comprises a reflective component model plus an open and extensible reconfiguration management infrastructure. Recently we have extended OpenRec to verify whether an intended (re)configuration would result in an application's structural constraints being satisfied. Consequently OpenRec can automatically veto proposed changes that would violate configuration constraints. This functionality has been realized by integrating OpenRec with the ALLOY Analyzer tool via a service-oriented architecture. ALLOY is a formal modelling notation which can be used to specify systems and associated constraints. In this paper, we present an overview of the OpenRec framework. In addition, we describe the application of ALLOY to modelling reconfigurable component based systems and highlight some interesting experiences with integrating OpenRec and the ALLOY Analyzer.

1. Introduction

Dynamic reconfiguration is the process of making changes to software systems at run-time. The motivation for dynamic reconfiguration spans many application domains and more fundamentally is driven by a need for either adaptive behaviour or very high availability. Adaptive software changes its behaviour at run-time in response to a volatile operating environment. Highly available systems need to execute in an uninterrupted manner for extended periods.

Mobile and ubiquitous systems are a widely cited [20, 22, 9] example of a class of system which requires adaptive behaviour. In particular, the execution environment of a handheld device is subject to fluctuation in resources that include power supply, network bandwidth, and available memory. Changing from a mains power supply to battery power might cause the software running on the device to make conservative use of the power supply. Similarly, the software could detect a move from a wired to a wireless network; in response to this the software might reconfigure itself to compress data before transmitting it to reduce the payload.

Systems with a requirement for high availability tend to be diverse but often mission-critical. A requirement of the Space Station's software control system, for example, is that it regulates oxygen to crew members. Clearly, this system cannot be shutdown for maintenance, without some other arrangements being made. Similarly, telecommunications switching systems cannot simply be shutdown as doing do so would withdraw critical functionality for handling emergency calls. Moreover, with many businesses adopting 24/7 modes of operation, there is a growing demand for high availability. A recent study [1] has revealed that outages are estimated to cost brokerage companies and banks US$4.5M and US$2.6M respectively. Regardless of the consequences of service unavailability, whether they endanger human life or be economic in nature, the challenge is to provide technology which enables software systems to necessarily evolve in order to remain useful [19], but to do so in a way that does not incur downtime as traditional maintenance processes do.

In recent years, a number of techniques have been developed which enable varying kinds of software reconfiguration. Primitive techniques include conditional statements and function pointers as mechanisms to develop "closed" adaptive systems, in which all adaptive behaviour is known at system design time [21]. More recently, Strategy objects and middleware interceptors have been used, in Dynamic TAO [26] and ACT [24] respectively, to support varying degrees of adaptation. To enable unanticipated run-time change, both for adaptive systems and for those with a requirement for continuous service, approaches based on reflection [7] and dynamic AOP [23, 25] look promising.

21st IEEE International Conference on Automated Software Engineering (ASE'06)
0-7695-2579-2/06 $20.00 © 2006 IEEE

These mechanisms promote openness and allow systems to acquire and offer new functionality and to change their non-functional characteristics in ways unforeseen at design time.

Regardless of the choice of mechanisms used to realize dynamic reconfiguration, there are a number of fundamental issues that need to be addressed when making run-time changes to a system. First, the presence of a dynamic reconfiguration capability should not compromise application *integrity/correctness*. In general, causing a system to fail or enter an erroneous state would negate any benefit brought about from dynamic reconfiguration. Second, the run-time *overhead* introduced by a reconfiguration management facility should be insignificant and acceptable. Third and finally, dynamic reconfiguration should be *transparent* to application developers. Non-transparent approaches introduce additional complexity and their correctness is dependent on contribution from application developers.

In our work, we are particularly concerned with preserving an application's integrity during periods of run-time change. OpenRec [13] is a framework for developing component-based reconfigurable applications. The framework is open and extensible with respect to reconfiguration management and allows a range of techniques for managing change to be implemented. In general, these techniques handle interactions between components during reconfiguration and ensure, for example, that components do not deadlock and that inter-component communications are not discarded. However, a significant limitation of OpenRec has been its lack of support for enforcing architectural constraints. For example, a configuration may be correct only if particular components are present or if there is a particular pattern of connectivity among the components.

During the past decade, formal modelling techniques have been applied to software architecture descriptions [8]. Early work by Abowd, Allen, Garlan and Shaw [2, 27] involved the use of the Z notation to formalize the computational data/state aspects of software architectures. However, automated verification support was lacking. Subsequently, Allen and Garlan [4] developed Wright, a CSP-like notation, to formalize the interactive communication aspects of software architectures. In this case, a translation tool was developed to generate CSP descriptions from Wright ADL, allowing the FDR model checker to be used to carry out consistency and completeness checking [3]. More recently, Georgiadis, Magee and Kramer [11] have used the ALLOY [15] language to specify constraints for Darwin architecture descriptions. However, detecting and recovering from constraint violations at run-time requires that code is manually prepared from the ALLOY specifications.

Our approach to modelling and enforcing an application's architectural constraints offers a more integrated and automated solution. Specifically, we have used ALLOY to specify architectural constraints. Moreover, we have integrated its tool support with OpenRec using a service-oriented architecture. At reconfiguration time, a proposed application configuration can be *automatically* verified with the result that the reconfiguration is either performed, in cases where all constraints will be preserved, or vetoed where it is detected that at least one constraint would be violated.

This paper is structured as follows. In Section 2, we present an overview of the OpenRec dynamic reconfiguration framework. In Section 3, we describe how ALLOY can be used to model both generic and application-specific constraints of component-based applications. In Section 4, we explain how we have integrated ALLOY Analyzer with OpenRec and report on some transferable experiences. Section 5 concludes the paper and identifies avenues for further work.

2. The OpenRec framework

OpenRec [13] has been developed to support automated reconfiguration of component-based applications. Figure 1 presents the framework's architecture and shows that it is organized by three layers: Change Driver, Reconfiguration Manager and Application.

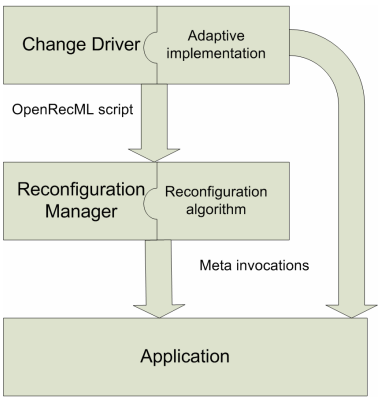

Figure 1. The OpenRec architecture.

For systems with a requirement for high availability, change is generally initiated by a third party. In this case, the Change Driver simply provides an interface enabling OpenRecML reconfiguration scripts to be submitted. OpenRecML is an XML-based language which is used to describe component configurations and alterations to existing configurations (i.e. reconfigurations). For adaptive applications, the Change Driver is a container which is intended to host a developer-provided implementation which determines *when* and *what* changes are required. In determining when to reconfigure, the Change Driver may use meta interfaces at the Application layer to dynamically insert monitoring code. For both highly available and adaptive sys-

tems the Change Driver outputs an OpenRecML script for processing by the Reconfiguration Manager.

The Reconfiguration Manager layer is also a container into which a particular implementation of a reconfiguration algorithm can be plugged. The algorithm determines how the reconfiguration will be managed so that a system's integrity will be preserved. The remaining layer, Application, is a configuration of applications components written by developers.

2.1. Reflective component model

Each of the layers is constructed using a reflective component model, described more fully in [13]. Components conform to Szyperski's definition of a component [28], being a unit of composition with well-defined interfaces. In addition to having provided interfaces, which specify the services offered to others, OpenRec components also have explicit required interfaces. Required interfaces make clear the services a component expects of others in order to satisfy its own contract.

Components are also interconnection independent, meaning that they have no knowledge of their connections with other components. To respond to an incoming communication, a component uses its provided interface without knowing which component actually initiated the communication. Similarly, to make an outgoing request, a component uses its required interface but does not know the component whose provided interface is bound to its own required interface. A configuration of components is thus a set of components where provided interfaces are bound to required interfaces. The loose coupling between components means that, in principle, the configuration can be changed by adding, removing, replacing and migrating components, and by changing the connectivity structure.

Being reflective, the component model provides two layers, a base layer and a meta layer which are causally connected [17]. With OpenRec, the base layer corresponds to application components and interfaces. The meta layer provides interfaces to discover and modify the configuration (cf. structural reflection) and to insert/remove interceptors on connectors (cf. behavioural reflection). Interceptors can be used for monitoring purposes and to introduce or withdraw arbitrary code which is executed during an inter-component communication.

To illustrate OpenRec and subsequent material on formalizing architecture, we introduce a simple case study which runs through the remainder of this paper. The system is a network of routers which carry messages from source nodes to destination nodes. Figure 2 shows a network router configuration. It involves a source node, two destination nodes and four intermediate router nodes.

Using OpenRec's component model, three Component

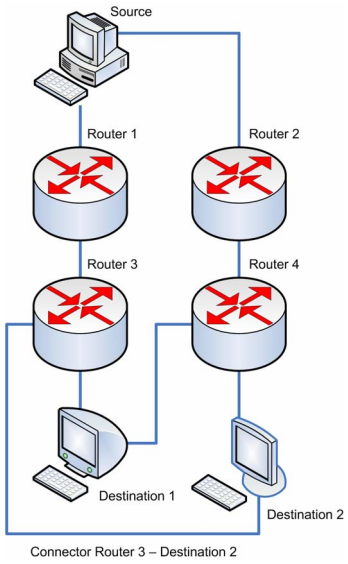

Figure 2. The Router component configuration.

subclasses have been developed: Source, Destination and Router. Source does not have any provided interfaces but defines one required interface, ISend, enabling Source instances to send messages. Similarly, Destination does not have any required interfaces but implements a IReceive provided interface allowing it to receive messages. Component Router both provides the ISend interface and requires the IReceive interface. Connectors in this application implement asynchronous messaging, essentially providing message queues.

Key non-functional requirements of the router application include availability and reliability. Informally, messages should be sent on demand with guarantees on delivery. To help meet these requirements, redundancy has been used so that there are (*at least*) two separate (distinct) communication paths between any Sender and Receiver nodes. Should an individual Router fail, a message can still be routed to its destination. For example, Figure 2 shows a configuration with the required redundancy, whereas if the connection between the Router 3 and the Destination 2 were removed, the configuration of Figure 2 would involve single points of failure.

2.2. Reconfiguration management

Using OpenRec's component model to develop an application offers rich possibilities for reconfiguration using the meta-level interfaces. However, arbitrary use of these interfaces can easily cause an application's integrity to be compromised. With regard to the router application, breaking connections (through disconnect()) with a Router and

removing it would cause a message to be lost if these actions were performed *after* the Router had taken a message from its input queue but *before* it had forwarded the message. More generally, this problem is referred to in the literature as synchronizing reconfiguration with the ongoing execution of the application [18].

Many approaches to synchronization have been proposed in the literature and include abstract reconfiguration protocols [18] where application components must be programmed to respond to reconfiguration commands[1]. Kramer and Magee's seminal work involves components communicating via transactions which can be multi-message exchanges between a pair of components. Other techniques have been developed for request-reply communication semantics and implemented as extensions to popular middleware, for example Java RMI [9] and CORBA [5]. In general these approaches involve establishing *a safe state* prior to and during reconfiguration where components operate in degraded mode. We refer the interested reader to [12] for a comprehensive review of synchronization techniques.

Part of the OpenRec framework is an interface named IReconfigurationAlgorithm which is intended to be implemented with a particular synchronization technique. This interface plays the role of strategy in the Strategy pattern [10]. The Reconfiguration Manager layer can be configured with any component providing the IReconfigurationAlgorithm interface. A basic implementation of the interface, ReconfigurationAlgorithm implements start() as a template method [10] which defers specific steps to subclass implementations:

- doCheckConstraints() gives an algorithm the opportunity to determine whether it is able to handle the reconfiguration. For example, Kramer and Magee's algorithm requires that a change management interface be implemented by components - and this can be determined using reflection.

- doSynchronise() should be implemented to carry out the specific actions of the algorithm to synchronize reconfiguration with the application. This typically involves calling change management operations on affected components or blocking interfaces to prevent a component from initiating or processing further communications.

- doReconfigure() is implemented to carry out the actual structural changes necessary to effect the reconfiguration. This step involves invoking the adaptation operations defined in the meta interfaces of application components.

[1]This is a case where a synchronization technique lacks transparency because contribution is required from application developers.

Many well-known synchronization techniques have been implemented in OpenRec. However, for the router application, we have developed a custom algorithm, based on [14] which incurs less overhead than a general purpose algorithm. Essentially, the algorithm delays reconfiguration of a Router component in cases where the Router has read a message until it has output the message to another Router or Destination node. The algorithm involves blocking a Router's provided interface to prevent it from reading further messages. Once blocked and it has output any expected message, the Router is deemed to have reached the safe state.

2.3. Discussion

OpenRec is inherently open and extensible with respect to the component model and reconfiguration management. With the former, new kinds of components and connectors with different semantics can be introduced. For reconfiguration management, new algorithms can be developed and made available at run-time.

OpenRec also promotes a healthy separation of concerns. Application developers are able to focus on application functionality separately to reconfiguration management. This is particularly true where reconfiguration algorithms are transparent, in other words where they need no contribution from developers. Developers may associate metadata with components to decorate them with information that algorithms may find useful. For example Chen [9] has proposed an algorithm which relies on knowledge of which operations of a component change its state. For adaptive systems, decision logic is encapsulated in the Change Manager and interceptors can be registered on connectors, all independently to core application logic.

Reconfiguration management is itself reconfigurable in OpenRec. The actual algorithm used by the Reconfiguration Manager layer can be substituted at run-time as it is simply a Strategy component. Adaptation logic can be changed at run-time since the Change Driver layer is also implemented using OpenRec's reflective component model. These features make possible some interesting capabilities such as self-learning and self-optimizing reconfiguration management.

However, based on the description of OpenRec in this section, it is clear that OpenRec is unable to enforce an application's structural constraints. Until recently, OpenRec has offered no way of expressing structural constraints or verifying them at reconfiguration time. Interestingly, we have been able to *dynamically* evolve OpenRec's dynamic reconfiguration management to offer this functionality - in a similar way that applications hosted by OpenRec can be reconfigured.

3. Formal constraint expression and verification

Traditionally, architecture descriptions are specified using diagrammatic notations or textual languages. However, configurations defined in this manner are likely to be inconsistent or error prone since there are no means of rigorously verifying their correctness. As a result, formal modelling techniques have been applied to the design, verification and development of more reliable and effective software architectures [2, 4]. The well-defined semantics and syntax of formal modelling languages make them suitable for precisely specifying and formally verifying architecture requirements. One of the most promising advantages in using mathematical and logic based techniques is that formal reasoning of the architecture properties can be achieved. Consequently, the correctness of architecture configurations can be guaranteed. In this section, we demonstrate formal modeling and verification of OpenRec architecture models using the ALLOY specification language and its tool support.

3.1. ALLOY and its Analyzer

ALLOY [15] is a structural modeling language based on first-order logic. It is suitable for expressing complex structural constraints and behaviors of software and hardware systems. The ALLOY language treats relations as first class citizens and uses relational composition as a powerful operator to combine various structured entities. The essential constructs of ALLOY are as follows:

- A **signature** ('sig' in ALLOY syntax) describes the properties of a set of entity objects. It introduces a given type, which consists of a collection of relations (called fields) and a set of predicates representing the constraints on the fields. A signature may extend fields and constraints from another signature.

- A **fact** ('fact' in ALLOY syntax) is a constraint on relations and objects that is always true within the specification. It is a formula that takes no arguments and does not need to be invoked explicitly.

- A **predicate** ('pred' in ALLOY syntax) is a template for a parameterized constraint. It can be applied elsewhere by instantiating the parameters. A predicate is always evaluated to either true or false.

- A **function** ('fun' in ALLOY syntax) is a template for a parameterized expression. It can be applied elsewhere by instantiating the parameters. A function evaluates to a value.

- An **assertion** ('assert' in ALLOY syntax) is a constraint that is intended to follow from the facts of a model. It is a formula whose correctness needs to be checked, assuming the facts in the model.

The ALLOY Analyzer [16] is an automated tool support for the ALLOY formal modeling language. It translates ALLOY specifications with a finite scope into prepositional formulas and generates instances that satisfy the properties expressed in the specifications by exploiting the SAT solvers. ALLOY Analyzer provides two kinds of automatic analysis, i.e., simulation and checking. Simulation refers to generating a snapshot of the system based on the facts or predicates defined in the model. Checking is preformed through attempting to generate a counterexample for an assertion. The former is good at showing the feasibility of a specification, where conflicting constraints could be detected. The latter is good at checking the correctness of a certain property in the system, where the assertion could be proved based on the facts defined in the model and within a finite scope of instants. When the ALLOY Analyzer succeeds in finding a solution to a formula, it produces both graphical and textual output of the solution.

3.2. Formalizing the OpenRec configurations

In this subsection, we first present a ALLOY formal definition of the OpenRec architecture description. Based on such a semantics, we demonstrate how the ALLOY Analyzer can be used to automatically verify OpenRec architecture configurations.

3.2.1 The OpenRec architecture style

An architecture description represents the high level structure of a system, which consists of the software components involved, the external visible properties of the components, and the communication patterns among the components [6]. We define the OpenRec architecture description in ALLOY as follows.

```
abstract sig Interface {}
abstract sig Component {
 provided : set Interface,
 required : set Interface
}
```

The above defines a typical component structure in the OpenRec architecture framework. It consists of a set of provided interfaces for offering services to other components; as well as a set of required interfaces for receiving services from other components in the system. The provided/required interface pattern describes the potential communications inside an architecture description. It is

when two components having a matching provided and required common interface pair, then a connection can be established between the components. A connection in the OpenRec framework is defined in ALLOY as follows.

```
abstract sig Connection {
 mapping: Component -> Component,
 interface: one Interface
}{
 one mapping
 no (mapping & iden)
 interface in (Component.~mapping).required
 interface in (Component.mapping).provided
}
```

The above states that a connection consists of a mapping from the source component to the target component through a common interface. The constraints define that there is only one such mapping in a connection, and a component can not connect to itself. Most importantly, the common interface for a connection must be both in the provided interface set of the target component as well as in the required interface set of the source component. Therefore, an OpenRec system configuration can be modelled in ALLOY as a set of components communicating through their set of connections.

```
abstract sig System {
 components : set Component,
 connections : set Connection
}
```

3.2.2 OpenRec configurations in ALLOY

Based on the above ALLOY architecture style description, the components in a specific system can be extended. For example, the network router components from Section 2 can be presented in ALLOY as follows.

```
one sig NetworkRoute extends Interface{}
abstract sig SrcNode extends Component{}{
 no provided
 required = NetworkRoute
}
abstract sig DestNode extends Component{}{
 provided = NetworkRoute
 no required
}
abstract sig Router extends Component{}{
 provided = NetworkRoute
 required = NetworkRoute
}
```

The above defines 3 types of components in the system: (1) source nodes that only transmit data, (2) destination nodes that only receive data and (3) routers that are capable of both receiving and transmitting data. Furthermore, specific system requirements (constraints) can be expressed as ALLOY assertions, such as:

```
assert noCircle {
 all s:System, rt:Router |
  (rt in s.components =>
   (rt !in rt.^(Connection.mapping)))
}
check noCircle
```

The above assertion states that *a router should not have outgoing data transmitted back to itself*. This is to prevent any circular structure in the network configurations. Let us consider the more reliable network structure (Figure 2), where the data sent from the source is almost certain to reach the destination. As introduced in Section 2, this requires that all network configurations *have at least two distinct paths between a source and a destination*. This property can be interpreted as *at least two paths should not have any routers in common*. Therefore, if an internal failure occurs in a router on one path, an alternative path can still be used to transmit data, which further ensures the fault tolerance property during a dynamic reconfiguration. We can define a ALLOY assertion to achieve this as follows.

```
assert multiPath {
 all s:System, src:SrcNode, dst:DestNode |
  (src+dst) in s.components =>
   (some cn:Connection, rt1,rt2:Router |
    getSource(cn) = src and (rt1+rt2) in
    src.(Connection.mapping) and
    (no ((rt1.^(Connection.mapping)-dst)
     & (rt2.^(Connection.mapping)-dst))))
}
check multiPath
```

The user expects these assertions (constraints) to be held across all configurations applied to the system. Suppose we have a network configuration defined in Figure 2 as follows.

```
one sig source extends SrcNode{}
one sig dest1,dest2 extends DestNode{}
one sig rt1,rt2,rt3,rt4 extends Router{}
one sig cn1 extends Connection{}{
 mapping = source -> rt2
 interface = NetworkRoute
}
one sig cn4 extends Connection{}{
 mapping = rt2 -> rt4
 interface = NetworkRoute
}
......
one sig RouterSystem extends System{}{
 components = source+dest1+dest2
              +rt1+rt2+rt3+rt4
 connections = cn1+cn2+cn3+cn4+cn5+cn6+cn7
}
```

Note that there are 7 connections in the model. By running the ALLOY Analyzer for this configuration, the assertions hold, which ensures that the configuration in Figure 2 is valid. In the case of a reconfiguration, for example, suppose the user modifies the system by removing the connection between Router 3 and Destination 2 and adding a connection between Router 3 and 4 in the above example, the ALLOY Analyzer will automatically detect there is a violation of the assertion *multiPath* in the model, and report an inconsistency (with a counter example). After careful inspection, we found that such a change will cause only one distinct path to exist between the Source node and the Destination 2 node, which indeed violates our original fault tolerance requirement - if Router 4 fails, there will be no data transmission from the Source to Destination 2. However, such an error could not be detected by the OpenRec framework without the formal verification support. This could result in an undesired system configuration that the user is not aware of.

4. Software integration through web service

In the previous section, we presented a formal approach to the verification of OpenRec architecture configurations. We found that the ALLOY Analyzer could provide functionality to validate models based upon user-constraints. Specifically it can be used to verify OpenRec (re)configurations prior to applying them to the system under execution.

OpenRec and ALLOY are clearly two useful tools that if combined would offer a powerful combination to realize structural constraint expression and automatic verification. However, OpenRec has been developed in Python in order to benefit from its support for rapid prototyping while ALLOY is a Java application. Integrating these tools thus requires the use of a mechanism which masks programming language heterogeneity.

4.1. Integration using a web service

Two obvious integration solutions include using a middleware platform such as CORBA or a more lightweight web-service mechanism. We have opted for the latter to avoid the relatively high software development costs associated with CORBA and to enjoy the deployment simplicity of a web-service, such as being able to use HTTP to avoid firewall difficulties. Furthermore, web service technology is sufficient as there is no requirement for CORBA services.

A web-service client component has been developed for OpenRec and registered dynamically as an interceptor on the connector between the Reconfiguration Manager and current algorithm component. The interceptor is thus able to intercept all start() requests which it does by introducing the new initial step of verifying the proposed configuration with respect to structural constraints. Depending on the outcome of verification, the interceptor either allows control to proceed to the algorithm component or vetoes the reconfiguration.

ALLOY provides a quite a complex API which we have simplified using the Facade design pattern [10]. The resulting interface, exposed as a web service, provides the following operations: *compile* an ALLOY model, *execute* a command on a compiled model, *iterate* over any solutions for the most recently executed command, *get* an XML representation of a solution, and *renew* a lease on the service. The service thus provides a generally reusable service which ca be used by any web-service client of which OpenRec is one. The service has been implemented using the Apache Axis toolkit.

The process of validating dynamic reconfigurations based upon a set of user constraints can be broken down into two key steps, (1) generating the model representing the specified system configuration, and (2) consuming the service to perform the validation using the generated model. A high-level overview of the process is shown in Figure 3.

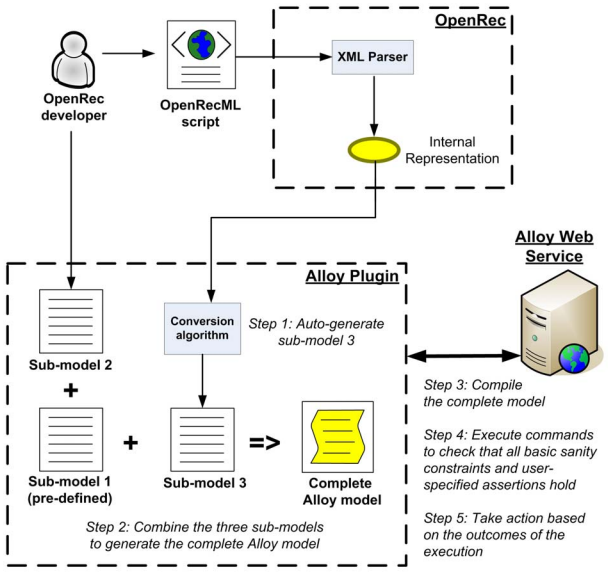

Figure 3. The system integration.

4.2. Generation of ALLOY models

As we mentioned earlier, in order to perform validation using the ALLOY web service, it is necessary to first create the ALLOY model that represents an OpenRec configuration. Upon each reconfiguration request, the corresponding OpenRecML script must be translated into an ALLOY model and sent to the web service for validation.

As we can see from Figure 3, an OpenRec configuration model can be divided into three sub-models, i.e., sub-model_1, sub-model_2 and sub-model_3. Sub-model_1 represents the high-level architecture style which is common to all OpenRec systems. Sub-model_2 is specified by the users and describes the specific components, interfaces, connections and constraints that are based upon the system being hosted on the OpenRec framework. User constraints are expressed as assertions such as in section 3.2.2. For example, in the network router, sub-model_2 contains definitions for 3 types of nodes and the connectivity constraints for source and destination nodes. Sub-model_3 specifies the information about a configuration itself. This is unique to each (re)configuration, but it can be automatically generated from the information in the OpenRecML script. An algorithm was developed to convert the OpenRecML elements into their corresponding ALLOY specifications. For example, consider the following segment of an OpenRecML script of the network example in Figure 2:

```
<add>
 <component
   uniquename="source"
   ...
 </component>
 <component
   uniquename="rt1"
   ...
 </component>
 <connection
   sourcename="source"
   targetname="rt1"
   interfaceclass="NetworkRoute"
   ...
 </connection>
</add>
```

From this the following ALLOY segment can be automatically generated:

```
one sig source extends SrcNode{}
one sig rt1 extends Router{}
one sig cn1 extends Connector{}{
 mapping = source -> rt1
 interface = NetworkRoute
}
```

Once an OpenRec configuration is successfully translated into a ALLOY representation, it can be verified.

4.3. Validating the router reconfiguration

As shown in Figure 3, step 1 and 2 involves automatically generating the complete ALLOY configuration model from a user input. Steps 3 and 4 are accomplished by using the ALLOY web service. It compiles the generated ALLOY model and executes each user-specified assertion to check whether the reconfiguration is valid. Step 5 involves taking actions based on the outcomes of the checking. For a configuration to be valid and to be applied by OpenRec, all the assertions within the model must pass. If any of the checks in step 4 fail, then the reconfiguration is deemed to be invalid. The OpenRec interceptor receives a counter-example and halts the reconfiguration being applied. For example, if an attempt is made to remove the connection between Routers 3 and Destination 2 and to add a connection between Router 3 and 4 from its original formation shown in Figure 2, OpenRec is able to detect the modification as invalid and prevents the system from being modified.

4.4. Performance overhead

The performance of the service was evaluated by comparing the times taken to invoke common operations through the ALLOY web service against those for ALLOY Analyzer. Only compile and execute command instructions were considered, and the results are summarised in Table 1. All measurements were obtained by calculating the average time to perform the operation 200 times. The process was repeated 5 times for each operation in order to reduce any random variations and to obtain reliable results. The same ALLOY model involving the router configuration was used across all the measurements.

Table 1. Average latency for remote and local invocations of compile and execute operations

	Average compile latency (ms)	Average execute latency (ms)
Local	86	2008
Remote	132	2129
overhead	46	121

4.5. Lessons learned

A basic design decision we made was to make the web service a *stateful* service such that the service retains state across operation invocations. The motivation for this was to reduce the run-time overhead at reconfiguration time incurred by interacting with the service. With a stateless implementation, every command executed by the service would require an ALLOY model to be sent as a parameter and compiled before executing the command. Not only would this involve unnecessary server processing but it would waste bandwidth too.

However, we faced three difficulties in developing a stateful web service. First and fundamentally, web-services

typically run over the stateless HTTP protocol which does not recognize the concept of session. Second, the service is intended for simultaneous use by multiple clients which requires that different state be maintained for each client. Third, as with any web-based application it is generally not known when the service has been finished with and hence when resources associated with a client can be freed.

The first problem was solved using Axis' service objects. A service object represents a connection and has one of three lifetimes: request, session or application. A request-type service object is created for every new incoming request, regardless of whether the request is from a client that has already made a recent request. A session-type service object is created for a particular client and reused when the client makes a subsequent invocation. An application-type object is a singleton shared by all clients. For our purposes, a service object of type session solves the stateless protocol problem.

The second problem, supporting concurrent clients, is not solved completely through the use of session-type service objects. ALLOY makes liberal use of static variables - for which there is one value per JVM. Hence, multiple clients sharing a common server JVM would likely conflict with each other by overwriting the values of static variables during interleaved calls. To solve this problem we employed one JVM per client. For each new client, a new JVM is launched and dedicated to that client, thus avoiding use of shared variables. The service object for a particular client delegates processing to the client's JVM using Java RMI. To simplify the process of spawning JVMs and interacting with them we used the ProActive library. The JVM-per-client solution is attractive in that it improves scalability and fault tolerance, particularly where the JVMs are physically distributed.

The solution to the third problem, determining when resources held for a client can be released, has been solved by a leasing technique. Periodically, the client proxy calls the renew operation of the web service. If a client fails to renew its lease, the server destroys the service object and JVM allocated to the client. This solution is more fault tolerant than the client making a finished call on the server since the call may never be made by a non-altruistic client or where a process, machine or network failure prevents the call from being received. This technique is commonly used for garbage collection in distributed object middleware.

One final difficulty we experienced was that ALLOY is multithreaded with some ALLOY API calls being non-blocking. Rather than return a result, these calls return immediately but cause an event to be subsequently fired. To hide this complexity, the Facade implementation made the calls, handled thread synchronization and event listening, and returned the response to the initiating service object.

5. Conclusions

In this paper we have described the addition to OpenRec of functionality which automatically verifies the structure of an application during periods of dynamic reconfiguration. OpenRec is a framework for managing reconfiguration of component-based applications, and is open and extensible with respect to reconfiguration management. The work described in this paper addresses a previous limitation of OpenRec in that until recently it has not been able to enforce an application's architectural constraints. Such constraints contribute to an application's integrity. Our approach has revolved around modelling constraints using the ALLOY language and using its tool support to verify them. To verify constraints automatically at reconfiguration time we have integrated OpenRec with ALLOY Analyzer using a service-oriented architecture.

In terms of established issues for dynamic reconfiguration, our work contributes to ensuring the *integrity* of applications as they are reconfigured. OpenRec delegates formal verification prior to reconfiguration to the remote ALLOY web service. While this incurs additional *overhead* (in the form of delay) *before* synchronising reconfiguration with the executing application, it does not interfere in any way with the application's operation. Only where all constraints are satisfied by a proposed reconfiguration does OpenRec proceed to actually carry out the reconfiguration. For adaptive systems, where the period between detecting the need for change and effecting the change often needs to be short, the additional overhead may be too costly. However, in more general cases where an evolution is necessary, but without urgency, the overhead of verifying an ALLOY model is more likely to be acceptable. Automated formal verification thus involves a trade-off between integrity preservation and overhead. With regard to *transparency*, our approach requires that developers contribute to one part of formal modelling, namely modelling the constraints specific to their applications.

Our experience with integrating two heterogeneous systems through a web-service involved a number of seemingly general problems, stemming from the stateful nature of the service. We used proprietary session-type service objects to provide a session for each distinct client. ALLOY's use of static (global) variables precludes simulanteous access by multiple clients and so a separate instance of the ALLOY application is spawned for each client. Although having each instance run as a heavyweight process consumes additional resources, instances can be physically distributed to improve performance and fault tolerance. To ensure server resources are released when no longer required, we used a leasing mechanism. To help expose an essential web-service interface, and to hide undesirable complexity, we applied the Facade design pattern.

In the future we plan to further investigate the relationship between model size and the time required for verification. To reduce verification time, one technique we are interested in pursuing is partitioning a model into smaller parts or sub-compositions that can be checked in parallel using a distributed computation service. In addition to exploring possibilities for developing a scalable approach for large systems, we are also interested in modelling other architectural characteristics such as QoS requirements. Finally, we are working on modelling other popular architectural styles so that applications which are expected to conform to these styles can be checked for actual conformance during reconfiguration.

References

[1] How much is an hour of downtime worth to you? Technical report, Must-Know Business Continuity Strategies, Yankee Group, Boston, July, 31 2002.

[2] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(4):319–364, 1995.

[3] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. CMU Technical Report CMU-CS-97-144.

[4] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*, 1997.

[5] J. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In *3rd International Symposium on Distributed Objects and Applications*, pages 197–207. IEEE Computer Society, September 2001.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 1998.

[7] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. M. Costa, H. A. Duran-Limon, T. Fitzpatrick, L. Johnston, R. S. Moreira, N. Parlavantzas, and K. B. Saikoski. The design and implementation of open orb 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[8] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM Press.

[9] X. Chen. Extending rmi to support dynamic reconfiguration of distributed systems. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 401, Washington, DC, USA, 2002. IEEE Computer Society.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[11] I. Georgiadis, J. Magee, and J. Kramer. Self-Organising Software Architectures for Distributed Systems. In *Proceedings of the first workshop on Self-healing systems*, pages 33–38. ACM Press, 2002.

[12] J. Hillman. *An Open Framework for Dynamic Reconfiguration*. PhD thesis, Lancaster University, United Kingdom, 2006.

[13] J. Hillman and I. Warren. An open framework for dynamic reconfiguration. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 594–603, Washington, DC, USA, 2004. IEEE Computer Society.

[14] J. Hillman and I. Warren. Quantitative Analysis of Dynamic Reconfiguration Algorithms. In *International Conference on Design, Analysis and Simulation of Distributed (DASD) Systems*, Virginia, USA, April 2004.

[15] D. Jackson. Alloy: A lightweight object modeling notation. Available: http://sdg.lcs.mis.edu/alcoa, 1999.

[16] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy Constraint Analyzer. In *The 22th International Conference on Software Engineering (ICSE'00)*, Ireland, June 2000.

[17] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[18] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[19] M. M. Lehman and L. A. Belady. *Program Evolution*. Academic Press, APIC Studies in Data Processing No 27, 1985.

[20] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[21] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[22] A. Rasche, M. Puhlmann, and A. Polze. Heterogeneous adaptive component-based applications with adaptive.net. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 418–425, Washington, DC, USA, 2005. IEEE Computer Society.

[23] A. Rasche, W. Schult, and A. Polze. Self-adaptive multi-threaded applications: a case for dynamic aspect weaving. In *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[24] S. M. Sadjadi and P. K. McKinley. Act: An adaptive corba template to support unanticipated adaptation. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 74–83, Washington, DC, USA, 2004. IEEE Computer Society.

[25] S. M. Sadjadi, P. K. McKinley, and B. H. C. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[26] D. C. Schmidt. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48, 2002.

[27] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[28] C. Szyperski. *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.